

# Reversi in C++

A summary of CMPT 310 final assignment

CMPT 310

**Quick Summary:** Pure Monte Carlo Tree Search is a superior algorithm to play Reversi given each turn has a limit of 5 seconds per turn. However, if this limit was removed and each available move were able to play the exact number of playouts, the Heuristic Pure Monte Carlo Tree Search seems to be a more superior one.

## Table of Content

### Section 1: Implementation Details

1A: Reversi Game Object.....	1
1B: AI Object.....	3
1C: Main/Testing files .....	4
1D: How to run.....	4
1E: How to play.....	4

### Section 2: AI Analysis

2A: Random Playouts.....	5
2B: Pure Monte Carlo Tree Search (PMCT).....	5
2C: Heuristic Monte Carlo Tree Search (HMCTS).....	6
2D: Testing Details .....	7
2E: Downfall of HMCTS.....	8
2F: Conclusion.....	9

<u>Reference</u> .....	10
------------------------	----

## **Section 1: Implementation Details**

### **1A: Reversi Game Object (reversi.cpp and reversi.h)**

This object contains only one class called board. Board is an 8 by 8 2d array that will contain the state of the game. In other words, it keeps track of which spaces are empty, occupied by black and occupied by white. To represent the state of the board in the array '\*' are used to represent an empty space, 'B' to represent that a black piece occupies that space, and 'W' to represent that a white piece occupies that space.

The object contains 2 constructors, one of which is a default one and the other takes in an 8 by 8 2d array. The default constructor will populate all 64 spots of the array (8 by 8) with '\*'. This is to represent a clean starting board. Meanwhile the constructor that takes in a 2D array will populate the board with the state described in that array.

Lastly the remaining 6 methods defined in the object are helper functions that will help with determining the state of the game.

- display(). This will print out the current state of the game as given in the board attribute.
- hint\_display(). This method takes in a pointer containing an array with all the valid move locations on a board. While printing out the state of the game from the board attribute, if the pointer states that spot is also a valid move location, we will mark on the display with an '!'.
- valid\_move(). This method takes in the color whose valid moves needs checking, along with the current turn. The array that gets returned is initialized to all 0's. For each 1's on the array it symbolizes a valid spot.
  - If the turn of the game is less than 5 it means the only valid moves that we have are the open spots in the centre of the board.
  - Otherwise, scan the board for empty spaces ('\*') and for each empty space that found, it will check its neighbours. As show in figure 1, the marked location around the '!' are its neighbours. Once a neighbour has been found that is a different piece color, we traverse in that direction until one of the following conditions.
    - If the edge of the board has been reached; this means that this spot may not be a valid move.
    - If it reached an empty spot on the board; this means that this spot may not be a valid move.
    - If it reached a piece that is of the same colour; this means that this spot is a valid move. The array will mark the initial spot with a 1.

- `input_move()`. This method takes in the color whose move we are going to place, and the (x,y) coordinates where we are going to place that move. The first thing this method does is to update the game board with the move at the specified coordinates.

Afterwards it does the following.

- At the specific board location that a move was just made scan all its neighbours. Prior to the calling of this function, the coordinates provided will be ensured a valid move. For each opposite piece colour found it will traverse in that direction until the following conditions have been reached.
  - If the edge of the board has been reached; It will go back and check another unchecked neighbour.
  - If it reached an empty spot on the board; It will go back and check another unchecked neighbour.
  - If it reached a piece that is of the same colour, it will start backtracking to the origin location. As it backtracks all pieces along the way will be flipped over to its colour. It will now check another unchecked neighbour.

As mentioned, since we verified the input to be valid prior to running this function it will guarantee a minimum of 1 flip per call.

Insert Figure 1...

- `get_black_score()` and `get_white_score()`. These methods will check the state of the game and retrieve the score of the respective colour. In other words, it will count each instance of 'B' and 'W' on the board.

## 1B: AI Object (ai.cpp and ai.h)

This object contains 2 methods one for each of the AI implemented. (More details about the AI go to section XXX)

- `pure_MCT()`. This method takes in the state of the board, number of playouts, a list of valid moves, a counter that contained the number of valid moves, the ai's piece color and the turn number. This is the implementation for the Pure Monte Carlo Tree search where we utilized an only the win, lost, draw statistics from random playouts.
  - Since each move should not take any longer than an average of 5 seconds, an artificial max run time was generated by dividing 5 by the total number of valid moves.
  - For each valid moves the AI had, random playout is simulated assuming the first move was a valid move. A function called `random_playout` is used which simulated games until it was over and return whether a win, lost or draw happened. The number of playouts has been set to 100000, however due to the size of the tree and a time cap was placed to simulate each valid move only a fraction of the playouts was done. (More on this in section 2B)
  - After either the time limit was reached or max playout have been played, we will check if the win percentage was higher than the current max win percentage. This win percentage was calculated by, the number of wins and draws divided by the number of plays. If it was high this valid move will overwrite the return variable with this move. Once all the valid moves have been simulated the move with the best win rate gets returned as the decision.
  - It will always return 1 move since the max score has been set to a low value that will never be reached by the calculation.
- `heuristicP_MCT()`. This method takes in the state of the board, number of playouts, a list of valid moves, a counter that contained the number of valid moves, the ai's piece color and the turn number. This is the playout for an AI using Pure Monte Carlo Tree Search with a few heuristics to help improve its decision making.
  - Like Pure MCT it does x number of playouts that gets either capped by time or the input called playout. However, it used a different random playout function called, `Hrandom_playout` which returns a struct called, `heuristic_return`. This struct will return whether it was a win, lose, or draw but also the number of available moves both the player and the AI had. This function also has a modified random playout which discourages both the player and the ai simulations to avoid unfavourable moves. (More to about this in section 2C)
  - After all playouts for a valid move has been completed a various number of heuristics are used to determine the best move. (More about this in section 2C)

- Like Pure MCT the move with the highest score will be selected.
- Both the AI have their turns to be limited approximately 5 seconds.

### **1C: Main/Testing (main.cpp, PvR.cpp, HvR.cpp, PvH.cpp)**

These files contain the bulk of the game logic as well as how we run tests to see which ai is better.

- main.cpp will be compiled to create the main application called Reversi. This is where you get to play against the choice of AI.
- PvR.cpp will be compiled to create one of the three testing application called PvR. This is where Pure MCT plays against a random choice algorithm. (One of the valid moves are randomly chosen.
- HvR.cpp will be compiled to create one of the three testing application called HvR. This is where the Heuristic PMCT will play against a random choice algorithm.
- PvH.cpp will be compiled to create one of the three testing application called PvH where the PMCT plays against the Heuristic PMCT.
- All the testing application have been set to play 10 consecutive games. (Takes a while if it takes 5 seconds per turn, with each game lasting approximately 60 turns.)
- In these files the main logic which determines if a game is over is also located inside.

### **1D: How to run**

If you do not want to recompile any code simply running “./x” will run the application. x is either “Reversi”, “PvR”, “PvH”, “HvR”. These are the application names that the makefile creates.

Using the provided makefile simply run “make clean” to remove all the “.o” files and application files created. Run “make” will generate the “.o” files along with the application files.

### **1E: How to play**

Each new game of Reversi you will be prompted with a series of questions such as which AI you would like to play against. Simply follow the prompts on the screen and you will make it into a game. The board starts blank but the only available move that you can make will be prompted to you. It will be obvious on the board where you can play (!) your next move and simply type it in the terminal where you want to make that move. When the game is over, you will be showing the final state of the game and be prompted if you would like to play again.

## **Section 2: AI analysis**

### **2A: Random Playouts**

Both the AI (Pure MCT and Heuristic PMCT) used roughly the same random playout algorithm. The only different is the Heuristic one asks the random playout to try and avoid picking the pieces in the x-square. The x-square are the squares diagonal to the corner pieces (A1, H1, A8, H8) which are very valuable places to place your piece. [1] The average playouts per seconds tends to improve drastically as more pieces gets played. The reason this is the case is that there are less moves required to reach the end state as more pieces gets played on the board. Starting with an empty board we roughly have 3300 playouts per seconds, but as we approach to the end state, we can have 21000 playouts per seconds. See figure 2.

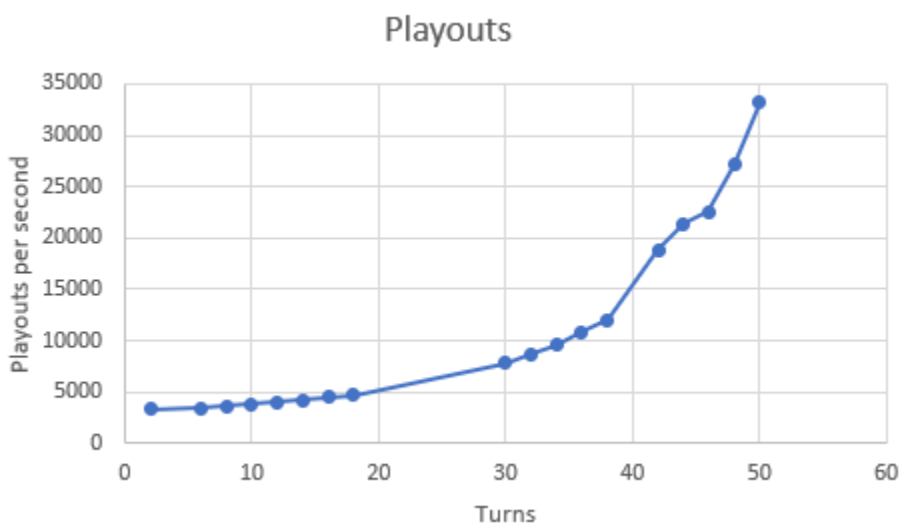


Figure 2: We can see that as the game progress (via turns) the number of playouts per second exponentially got faster. We the same result when running the random playouts for the Heuristic PMCT.

### **2B: Pure Monte Carlo Tree Search (PMCT)**

This algorithm run random playouts in order to determine which move is the best. Since we must make a move within 5 seconds, for each valid move that it can make the amount of time it we simulate the random moves is 5 sec/total moves. This means if we have 5 moves for a given state, then each move gets simulated for a maximum of 1 second. In that 1 second it will try to simulate as many playouts as possible. At the end it will use the win, lose, draw ratio to determine which is the best move. A variety of combinations were tested but, in the end, a simple “win + draw” combination resulted the best result again random choice algorithm. (This is our main benchmark to determine how good the algorithm played. After 100 games, 50 of

which PMCT went first, and the other 50 it went second, PMCT had an impressive 100 %-win rate against the random choice algorithm.

## 2C: Heuristic Pure Monte Carlo Tree Search (HPMCT)

This algorithm uses random playouts as a baseline to determine which move should get an edge. Also, this algorithm tries to ensure both the player and it plays the “best” possible moves. This is simply by avoiding placing moves in the x-square as possible. This is bad as placing a move in the x-square early can easily lead to a loss in a corner.[1] Which can skew the data to more losses than necessary. We also encouraged corner moves if that was possible. The final score of a move is calculated by the following formula: [2][3]

$$Score = win\% + position + mobility + inner - greedy$$

Overall, this formula is a mobility-based calculation which will choose to play “bad” moves early on if it maximized its available move the most. By mid to late stages of the game that is when this algorithm will choose to flip as many pieces into their favour.

Win%: is simply a percentage that was recorded from random playouts.

Position Score: is a position score that tries to encourage corner moves, a-square and b-square (edge area in between c-squares) [1] and discourages c-square (squares adjacent to corners) and x-square.

### Notation





	a	b	c	d	e	f	g	h	
1			C	A	B	B	A	C	1
2	C	X					X	C	2
3	A								A
4	B							B	4
5	B							B	5
6	A							A	6
7	C	X					X	C	7
8		C	A	B	B	A	C		8
	a	b	c	d	e	f	g	h	

Figure 3: Image from [1]. The squares marked C are the c-square, X are the x-squares, A are the a-square, B are the b-squares.

Mobility: Calculated by taking the following equation:

$$mobility = \sqrt{\frac{ai_{mobility} - player_{mobility}}{ai_{mobility} + player_{mobility}}}$$

$ai_{mobility}$  is the total available moves the ai had when it did the random playouts.

$Player_{mobility}$  is the total available moves the player had when it did the random playouts.



The move mobility the Ai the more choice that it had to flip the board it is favour. However, this is also a downfall given the restriction of this heuristic.

Inner: is a heuristic that slightly pushes the algorithm to play in the center as much as possible. This allows a slight bias to move in the center as opposed to those out in the side. (However only important in the first 20 stages of the game.) Prevents the game space from being too open.

Greedy: is a penalty for the ai if it a move flip more states early in the game. Unless this move wins the game, there is no need to make this play. Flipping more states will lower out overall choices thus lowering our mobility greatly.

Despite having all these wonderful heuristics to help with the decision of the best move, this heuristic did not have a 100%-win rate against random choice algorithm. After 100 games, 50 of which it went first and 50 of which it went second, we only achieved a 75%-win rate against the random choice algorithm. There were slightly more losses when it went first but it is with in reason that turn order did not matter.

## 2D Testing Details:

Average game time for these were about 2.5 mins each as the AI needed 5 second per move while random was instantaneous.

	FIRST MOVE (BLACK)	SECOND MOVE (WHITE)
<b>PURE MONTE CARLO TREE SEARCH</b>	50 Win 0 Loss 0 Draw	50 Wins 0 Loss 0 Draw
<b>RANDOM CHOICE VS. PMCT</b>	0 Win 50 Loss 0 Draw	0 Wins 50 Loss 0 Draw
<b>HEURISTIC PMCT (HPMCT)</b>	36 Win 10 Loss 4 Draw	39 Win 9 Loss 2 Draw
<b>RANDOM CHOICE VS. HPMCT</b>	10 Win 36 Loss 4 Draw	9 Win 39 Loss 2 Draw

Table 1: This is the win, loss, draw it had versus random choice.

When we pitted PMCT and HPMCT together, we can see that PMCT was better with about a 60%-win rate after 50 games. (Average game time took about 5 mins)

	FIRST MOVE (BLACK)	SECOND MOVE (WHITE)
<b>PURE MONTE CARLO TREE SEARCH</b>	12 Win 10 Loss 3 Draw	16 Win 5 Loss 4 Draw
<b>HERUISTIC PMCT (HPMCT)</b>	10 Win 12 Loss 3 Draw	5 Win 16 Loss 4 Draw

Table 2: This is the win, lost, draw for HPMCT vs. PMCT. We can see PMCT is superior compared to HPMCT.

However, if we allowed HPMCT more playout time (such that each moves gets approximately 1 seconds to do its playout from running this test for 20 games with random turn order (who goes first) We have an increase of HPMCT with an 70% win rate vs. Pure MCT.

	WIN RATE (RANDOM FIRST)
PURE MONTE CARLO TREE SEARCH	4 Win 14 Loss 2 Draw
HEURISTIC PMCT (HPMCT)	14 Win 4 Loss 2 Draw

## 2E: Downfall of HPMCT

One of the goals that my heuristic tries to do early on is to minimize the number of flips, as well as maximizes its available next move. By making these choices as we progress further into the game, HPMCT will have on average 20 available moves by turn 30 while the player will have about 5 to 10 moves. Near the end of the game, HPMCT will make massive gains in the point total due to how flexible it is. However, this came at a cost, where HPMCT could easily lose if bad moves were chosen.

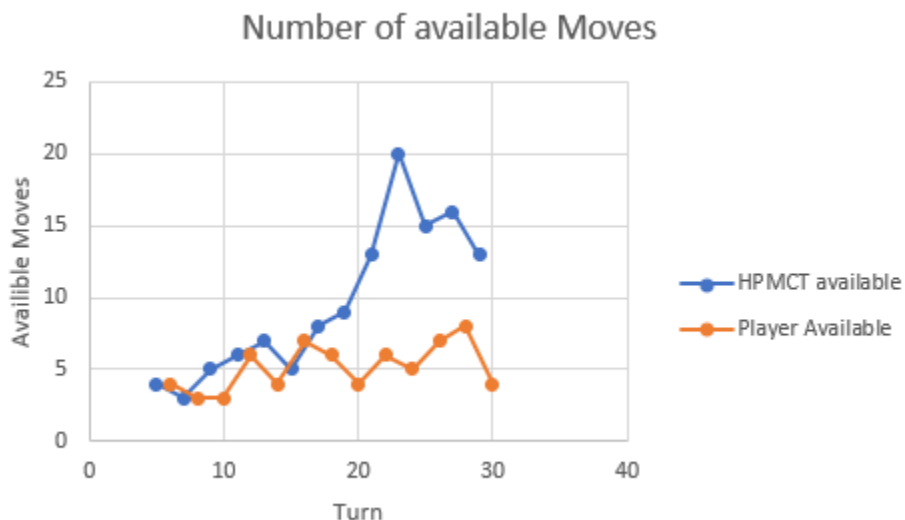


Figure 4: Shows us how many available move that HPMCT had compared to the opponent.

While having many available moves around turn 30 gives us more chances to make a comeback this is where the heuristic's biggest downfall lie. Since each heuristic is limited to a maximum of 5 seconds per turn, the more available moves we have the less available time we get to do those playouts. For example, around turn 22 HPMCT has about 20 available moves that it can play. Thus only  $0.25(5/20)$  seconds are allocated for each available moves' playout.



Figure 5: As more available moves are opened; we see max time decreasing

Looking back to figure 2 at 22 turns we can see that in a second roughly 7000 playout can be played each second.

Thus, on average each move gets approximately 1750 playouts. Due to

the nature of random playouts the resulting simulations would not be reflective on which move is the best choice given the low playouts. Also due to the high number of available moves, many moves might be bad ones that would greatly affect the rest of the game or set HPMCT in a bad position. As a result, if we gave all available moves exactly 1 second of playout times, we were able to greatly see a high win percentage for it. However, is not a feasible option to fix HPMCT as this would means some turn would end up taking 20 plus seconds to determine a move from HPMCT.

## 2F: Conclusion

I attempted to make a heuristic approach with pure Monte Carlo Tree search but unfortunately it was unable best the pure Monte Carlo Tree search that it was based out of. The main reasons for failure was time constraints, due to how the heuristic approach took a losing stance to maximize its mobility (number of available turns) early on in order to make calculated plays to make massive gains. During the time it was maximizing its mobility, it generally chooses moves that but it in a worse position due to the nature of random playouts. If more plays were made, statistically the bad moves would have been weeded out. Due to the lack of plays this was not the case.

### References:

<sup>1</sup> "Strategy Guide for Reversi & Reverse Reversi." Samsoft.

<https://web.archive.org/web/20200206234321/http://samsoft.org.uk/reversi/strategy.htm>

(accessed on Aug 9th 2020) (website was down when writing the references.)

<sup>2</sup> "Need Heuristic Function for Reversi(Othello) ideas." Stackoverflow.

<https://stackoverflow.com/questions/13314288/need-heuristic-function-for-reversiothello-ideas>

<sup>3</sup>P. Parek et al. "Othello/Reversi using Game Theory techniques." Play-Othello. <http://play-othello.appspot.com/files/Othello.pdf>